ATTACK ON RSA BY BRANCH PREDICTION

INPG-UJF / CSCI Master-1
Intensive Project in Security

Hamad Raeisi
hamad.raeisi@ensimag.imag.fr

Sultan AlTamimi
sultan.altamimi@ensimag.imag.fr

Abstract

Simple Branch prediction analysis attack, a type of a side-channel attack, allows an unauthorized spy process to collect information from the victim process. Side-channel attacks are based on leakage in the physical implementation of a system. Branching conditions can provide information on sensitive data handled by the processor, and since the Branch Prediction Unit (BPU) is shared by all the processes running on a processor it is possible for the spy process to collect useful information of another process.

In this paper, it is shown that a spy process running simultaneously with an RSA process is able to extract all the secret key bits during one single RSA decryption execution. This attack was performed on the OpenSSL (version 0.9.8g) implementation of RSA. In order for this attack to work, first it is necessary to be able to run the spy process on the same processor of the victim processes. Then make sure the processor has an interface to monitor, in details, all branches made within a range of addresses and lastly to be able to compile and execute a newly written program.

The leakage of such crucial branching information directly depending on the private key has been stopped in later versions of OpenSSL.

Keywords: Side Channel Attacks, Simple Branch Prediction Analysis, OpenSSL, RSA, Square and Multiply Algorithm.

1. Introduction

Side channel attacks are techniques in which an attacker can extract secret information leaked from the implementation of a cryptosystem. The most common types of this attack that are measurable from the execution of the RSA algorithm are the timing attacks, power consumption and Simple Branch Prediction Analysis (SBPA) attack. The SBPA attack takes advantage of side channel information leaked by the branch prediction unit (BPU). The implementation of RSA on OpenSSL that is attacked uses the square and multiply algorithm, with no optimizations, in order to carry out the modular exponentiation. The attack exploits the conditional branch in the algorithm that depends on the bits of the secret key. So, by using a spy process that runs simultaneously with the victim process which runs an RSA decryption, the bits of the private key can be retrieved.

The report is divided into five main sections. The first section gives a theoretical overview about RSA cryptosystem and branch prediction. The second section describes the SBPA attack method and the tools used for the attack. The third section illustrates and analyses the results obtained from performing the attack. The fourth section shows some countermeasures that can be used to prevent the attack, and the last section gives a brief conclusion about the report.

2. Background Information

The goal of this section is to illustrate the theoretical aspects of the project. It explains the RSA cryptosystem and how it works. Furthermore, it gives explanation about branch prediction.

2.1 RSA Cryptosystem

RSA cryptosystem is one of the most well-known public key cryptosystems in the world. It was invented by Rivest, Shamir, and Adleman in 1977 [1].It can be used both for public key encryption and digital signatures. The main advantage of public key cryptography is that it solves the problem of all prior cryptography in which a secure channel is established for exchanging the key. On the other hand, the major disadvantage of the algorithm is that it requires a key size of at least 1024 bits for good security. Thus, it makes the algorithm quite slow. The security of the algorithm is based on the difficulty of factoring large integers.

2.1.1 Key Generation

The RSA algorithm starts first by generating two keys namely the public key and the private key. It uses computations in Z_n, where n is the product of two different prime integers p and q. The totient of the integer n is computed by applying Euler's totient function $\varphi(n) = (p\text{-}1)(q\text{-}1)$.

The Euler's totient function determines the number of positive integers less than or equal to n that are coprime to n. The encryption exponent is generated by selecting an integer e such that $\gcd(e, \varphi(n))=1$. The decryption exponent d is computed such that $d = e^{-1} \bmod \varphi(n)$. Consequently, the public key is (n, e) and the private key is (n, d). The values of d, p, q and $\varphi(n)$ are kept secret.

2.1.2 Encryption

In order to send an encrypted message, the sender A does the following:

a. Obtain the recipient B's public key (n, e).
b. Represent the plaintext message as a positive integer m.
c. Compute the ciphertext $c = m^e \bmod n$.
d. Send the ciphertext c to B.

2.1.3 Decryption

The recipient B does the following to recover the plaintext:

a. Uses his private key (n, d) to compute $m = c^d \bmod n$.
b. Extracts the plaintext from the message representative m.

2.1.4 Square and Multiply Algorithm

Both the encryption and the decryption operations in the RSA cryptosystem are modular exponentiation operations. This operation takes the form of $x^y \bmod n$ and the computation can be done using y-1 modular multiplications; nevertheless it is inefficient in case of large y. Accordingly, *square and multiply algorithm* reduces the number of modular multiplications required to calculate $x^y \bmod n$ to at most $2l$, where l represents the number of bits in the binary representation of y.

The square and multiply algorithm assumes that the exponent y in represented in binary notation such that [1]:

$$y = \sum_{i=0}^{l-1} y_i 2^i$$

where $y_i = 0$ or 1, $0 \le i \le l$-1. The algorithm to compute $z = x^y \bmod n$ is shown in Figure 1.

$z \leftarrow 1$

for $i \leftarrow l$ -1 **downto** 0

do
- $z \leftarrow z^2 \bmod n$
- **if** $y_i = 1$
- **then** $z \leftarrow (z \times x) \bmod n$

return (z)

Figure 1: Square and Multiply Algorithm

2.2 Branch Prediction

A branch instruction is a point in the instruction stream of a program where the next instruction is not necessarily the next sequential one. There are two types of branch instructions, unconditional instructions such as procedure calls and goto, and conditional instructions like if-then-else and for loops. In modern processors, branch predictor is used to determine whether a conditional branch in the instruction flow of a program is likely to be taken or not. This operation is called *branch prediction*. Branch predictors are used to keep the pipeline full of instructions. Thus, it enhances the performance and allows useful work to be completed while waiting for the branch to resolve [2].

In case of conditional branches, the decision to take the branch or not to take it depends indeed on the condition in which it must be evaluated. During this evaluation period, the processor executes instructions from one of the possible execution paths in which it saves time. Therefore, a branch prediction algorithm is used in order to predict the most likely execution path in a branch. If the predictor makes correct prediction, the program will continue normally without any delay. On the other hand, if the prediction fails (misprediction), the instructions on the pipeline must be flushed and discarded. In this case, the execution starts over from the mispredicted path and there will be a delay.

The processor requires mainly two kinds of information in order to execute the branches, which are the outcome of the branch and the target address of the branch. Branch Prediction Unit (BPU) of the

processor handles the prediction process. As shown in Figure 2, it consists of two main parts, the predictor and the Branch Target Buffer (BTB).

The predictor is used to predict the outcome of the branch in which it can be either taken or not taken. This prediction is based on the history of the same branch as well as the history of other branches executed just before the current branch. On the other hand, the BTB is used to store the target address of previously executed branches. That is each time a conditional branch is evaluated, the target address is recorded in the BTB for future use. Accordingly, when the prediction turns out to be taken, the instructions in the target address have to be fetched and issued [3].

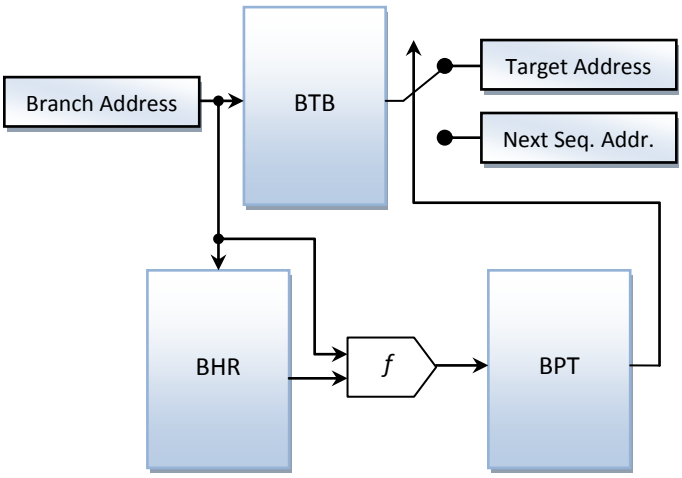

Figure 2: Branch Prediction Unit Architecture

3. Description of the Attack

This section is divided into two subsections. The first section describes the Simple Branch Prediction Analysis (SBPA) attack and how such a technique can help to retrieve crucial private information. The second section describes the victim and spy process used. OpenSSL is the victim process while Pfmon is the spy process which will help retrieve the private key.

3.1 Simple Branch Prediction Analysis (SBPA) Attack

SBPA is a type of a side channel attack. Side channel attacks are attacks that exist due to the poor implementation of a system. However, the model system can be proved to be very secure but the implementations of such system will make it vulnerable to some attacks. Those types of attacks require a lot of technical knowledge about the systems internal operations [4].

Branch prediction is a feature that can be found on almost all the latest processors. It makes the processer runs at high speed by keeping the pipeline full. A special Branch Prediction unit (BPU) in the processor uses a highly optimized branch prediction algorithm to predict the direction and outcome of the instructions being executed through multiple levels of branches, calls and returns. The prediction of the instruction helps the instruction to be executed with no waiting.

The attack to be used here is based on branch prediction analysis and also referred to as "Trace-driven Attack against the BTB". The victim process usually will execute the RSA processes by using one of the exponentiation algorithms (usually it is the square and multiply exponentiation algorithm) which will contain a branching event on the number of bits in the private key *d.* On the other hand, the spy process which is running simultaneously will be monitoring the number of branches and the decision taken at each branch. This data gathered will help in extracting the private key [5].

The spy process should be started with the victim process. The spy processes will fill the BTB with it conditional branches and measures the overall execution time of all its branches. All those branches will be mapped to the same BTB set which also stores the specific conditional branches for the private key bits. Therefore to find all the bits of the private key it will be needed to execute only spy branches and measure their overall execution time. Therefore, the spy process will see the complete Taken/Not Taken trace of the target branch and is able to retrieve the private key.

3.2 Tools used for the attack

Two different tools are used in the process of attacking RSA by branch prediction. The first tool which is OpenSSL is a tool which implements the RSA process which is to be attacked. The other tool which is Pfmon, is a tool used to monitor the branches made spy process in order to retrieve the private key.

3.2.1 OpenSSL

The OpenSSL is a certificate management tool with shared libraries that provide various encryption and

decryption algorithms and protocols, including DES, RC4, RSA and SSL. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. The OpenSSL version used in this project is 0.9.8g which is vulnerable to the attack performed [6].

The RSA library of OpenSSL is the library used in this project. This library uses another library to manage large integers which is called BIGNUM. Standard libraries in C allow the use of limited size integers. RSA requires to use very large integers in order to make the process secure.

A lot of functions are implemented for the encryption and decryption process of RSA. The function of our interest will be the function which performs the modular exponentiation. These kinds of operations are performed by very efficient algorithms to help achieve better performance.

In OpenSSL, the function that performs the modular exponentiation is called BN_mod_exp_mont. This function performs modular exponentiation for BIGNUM Montgomery. Montgomery reduction is an algorithm introduced in 1985 by Peter Montgomery that allows modular arithmetic to be performed efficiently when the modulus is large. Montgomery is a method that reduces multiplication, but it has no influence on the attack operation [7].

OpenSSL's RSA also uses the sliding windows technique which is an optimization on the algorithm by storing values in a table. This technique increases the size of the window based on the number of bits. Initially the window size is set to one, then if the exponent of d is larger than 23 bits it will set it to 3, if d is more than 79 bits it will set it to 4, if d is greater than 239 bits it will set it to 5 and if it was more than 671 bits it will set it to 6.

In the case where the window size is 1, it is clear that there is special treatment in case there is a bit set to 1 in the exponent *d*, which is a conditional branching to avoid an additional multiplication. The condition of a test connection is made directly on a bit of the private key. If we can recover following the evaluation of this algorithm conditions, then we can reconstruct the private key d.

3.2.2 Pfmon

Pfmon is tool used to monitor the performance of selected events on the processor. It can be used to count simple events or samples from bounded address ranges or an entire system. It makes full use of the libpfm library to help in programming the PMU. The processor that is best to be monitored with this tool is the Itanium 2 processor, which is the processor used in this project. This architecture has 16 PMU configuration registers (PMC), 18 data registers (PMD) and 4 other counters. In this architecture, the BTB cannot contain more than 4 pairs of addresses. To analyze the content, 9 registers are needed, from those 9 registers 8 to contain the addresses and 1 extra to keep a pointer to the last entry in the BTB [8].

In order to monitor the processor and trace the branches that were executed the event that is to be monitored is BRANCH_EVENT**.** This event alone is not sufficient. Some other options are to be used with this event. Another option that should be specified is --long-smpl-periods which will set long sampling periods for each event to display more info about each event. This will show details if the branch is taken or not. Having those two options indicated will give information on a large number of branches performed. Therefore another option to be used is the --irange option. This option will allow specifying the range of addresses to be monitored. The range of addresses to be monitored us found by debugging the program to be monitored.

4. Analysis and Results

The aim of this section is to discuss and analyze the results obtained from the experiments done during the project. There are two experiments, where the first one is exploited to demonstrate the attack using a simple program, and the other one is used to illustrate the attack on RSA implementation of OpenSSL. The attack is mainly attempted using Itanium 2 processor. But, it is also tried on the Intel Core-based processor.

4.1 Emphasizing the attack using a simple program

A small function called *doloop* as shown in Figure 3 is implemented in order to emphasize the attack using pfmon tool. Pfmon works as a spy process in which it starts before the victim process (doloop) and fills the BTB with its conditional branches addresses. In addition, it continuously executes its branches and measures their overall execution time. On the other hand, when the victim process starts and if it has a conditional branch to be executed, the BTB will be modified. The spy process will measure the overall

execution time of the branches and it will observe the changes in the BTB. Therefore, it can detect that the victim conditional branch is taken.

```
#define L 5
void doloop(int N) {
    int a , j;
    for ( j=1; j<=N; j++)
        if (j%L==0)
            a=1;
  }
```

Figure 3: Simple Program

Pfmon is used to trace the branch and observe whether it is taken (1) or not taken (0). A debugger (GDB) is used to obtain the assembly code of the conditional branch instruction as shown in Figure 4 by disassembling the doloop function.

```
0x4000000000000980 <doloop+272>:
[MFB] nop.m 0x0
0x4000000000000981 <doloop+273>:
nop.f 0x0
0x4000000000000982 <doloop+274>:
(p06) br.cond.dptk.few
0x40000000000009b0  <doloop+320>
0x4000000000000990 <doloop+288>:
[MFI] adds r15=-12,r2
```

Figure 4: Address Range of Conditional Branch Instruction

As can be seen from the previous figure, the target branch has the address 0x40000000000009b0. Therefore, when the condition (if (j%L==0)) is false, this target branch is taken and vice versa.

By using the pfmon option (--irange) which specifies an instruction address range constraint and start the simple program with N=10, pfmon is run as follows:

```
$ pfmon --long-smpl-periods=1 --smpl-entries=100 -e
BRANCH_EVENT --irange=0x4000000000000980-
0x4000000000000990 -- ./loop 10
```

The output stream of pfmon is analyzed according to the value associated to "taken". Thus, if taken=y, then the bit of the reconstructed sequence equals to 1 and vice versa. A sample of the pfmon output sequence is provided in the appendix A. A small program called *analyze* is used to recover the sequence directly from the output stream of pfmon. It looks for the values associated with the string "taken". For instance, if taken=y, it produces 1. As a result, the reconstructed sequence when N=10 is 1111011110.

It can be noticed from the reconstructed sequence that there are two zero bits. This means that the condition (if (j%L==0)) is satisfied and the branch is not taken at those two iterations. Thus, this trick can be used to determine the value of L.

Consequently, it is possible to recover the bits of the private exponent d of RSA by tracing the execution of the conditional branch instruction in square and multiply algorithm while decrypting the ciphertext.

4.2 Attacking RSA Implementation of OpenSSL

This section will elaborate on how the attack is performed on the RSA implementation of OpenSSL. First of all, a program should be written to decrypt an encrypted RSA message, this program is the victim program and pfmon has to monitor it while running. This program should be complied with a modified version of OpenSSL where the window size is always set to 1 and BN_FLG_CONSTTIME to 0 and compile OpenSSL with no optimization (-O0). BN_FLG_CONSTTIME is a flag and setting it causes BN_mod_exp_mont() to use the alternative implementation in BN_mod_exp_mont_consttime(). Therefore, the message will be decrypted without exposing the private key which will not allow the spy process to recover the private key.

This program will take as inputs n, d and e. a test program was run with the following keys:

n=
e02137497237346fbf66c1c92005adf442fa23026e4c
717da5950fce2af67433a7126fcd6738b5a8a7983c22
1a1161f6c65067e1296a08f81d3c93e2651c91ad
e=10001
d=
491e0cef44f7857fbf2d42a2de737be067c93a8a9c79
0bbd35bb7f407efb8fc47d7e73900292eac6d8d72dfe
277e73fe4340189ad153062bbbd8f4703d531f81

The key used in this program is of size 512 bits. The key was created by OpenSSL and will be used in the victim program to decrypt a chosen message from which the spy process will analyze and retrieve the private key without having any information about any data used in the program.

The target process is a program which calls the RSA_private_decrypt function of the OpenSSL library this function does the decryption. This function undergoes several different steps for decryption and the most crucial step is the modular exponentiation step. This step will allow recovering the key. So, by debugging the program it is noticed that modular exponentiation is done in a function called BN_mod_exp_mont (in the file bn_exp.c available with the OpenSSL library, see appendix B). The critical if condition (branch event) is the if condition with the following condition if (BN_is_bit_set(p, wstart) == 0). Only the branches of this "if condition" should be monitored. Therefore, the spy process should monitor only the addresses of this "if condition" which can be achieved by running the program in debugging mode. If the jump is taken, the multiplication is not done; therefore it detects a 0, and if the jump is not taken 1 is detected. The figure below shows the assembly code where the branching occurs. The branching occurs in the address which contains br.cond.dptk.few. This instruction means that there is a branching condition if the prediction is taken.

```
0x400000000007d4e0 <BN_mod_exp_mont+2464>:
        [MII]       mov r1=r42
0x400000000007d4e1 <BN_mod_exp_mont+2465>:
        mov r14=r8;;
0x400000000007d4e2 <BN_mod_exp_mont+2466>:
        cmp4.eq p7,p6=0,r14
0x400000000007d4f0 <BN_mod_exp_mont+2480>:
        [MFB]       nop.m 0x0
0x400000000007d4f1 <BN_mod_exp_mont+2481>:
        nop.f 0x0
0x400000000007d4f2 <BN_mod_exp_mont+2482>:
        (p06) br.cond.dptk.few
        0x400000000007d5d0
        <BN_mod_exp_mont+2704>
0x400000000007d500 <BN_mod_exp_mont+2496>:
        [MMI]       adds r14=-352,r41;;
```

Figure 5: Branching Assembly Code

The following is the command which is executed to run the spy process:

```
$ pfmon --long-smpl-periods=1 --smpl-entries=10 -e
BRANCH_EVENT --irange=0x400000000007d4f0-
0x400000000007d500 -- ./attack > attack_output
```

This function will give us details for each entry if it was taken or not, therefore a small program was implemented to analyze those results and display results in binary and hex, the following is the command used:

```
$ ./analyze attack_output | ./bin_hex
```

The analysis will analyze the data and produce a binary output as follows:

```
10010010001111000001100111011110100010011
10111100001010111111101111110010110101000
01010100010110111100111001101111011111000
00110011111001001001110101000101010011100
11110010000101110111101001101011011101101
11111010000000011111011111011100011111100
10001111101011111100111001110010000000001
01001001011101010110001101101100011010111
01011011111111000100111011111100111001111
11110010000110100000000011000100110101101
00101010011000001100010101110110111011000
01111010001110000001111010101001100011111
0000001
```

This binary represents the private key which is then converted to hex by running the program *bin_hex* and the output in hex is as the following:

```
491e0cef44f7857fbf2d42a2de737be067c93a8a9c79
0bbd35bb7f407efb8fc47d7e73900292eac6d8d72dfe
277e73fe4340189ad153062bbbd8f4703d531f81
```

As we can see that the output is identical to the private key *d* shown earlier which indeed shows the effectiveness of this attack.

4.3 Analyzing the attack on Intel core-based processor

The attack is attempted on a processor other than the Itanium 2. In this case also, the simple program is used to emphasize the attack and the private key of the RSA implementation is tried to be retrieved.

The pfmon tool in this case can be used to give the number of the taken branches. Nevertheless, it does not give such detailed information about the branches as in the case of Itanium 2 processor. So, by using a small trick in which the simple program is run N+1 times in order to observe at each iteration that the conditional branch is taken or not. For example when N=10, the program is run 11 times starting with N=0 and ending with N=10. At each two consecutive executions, if the number of branches is incremented by 2, this means that the branch is not taken;

however, if it is incremented by 1, then the branch is taken. pfmon is executed as the following:

```
for i in $(seq 0 10) ; do pfmon -trigger-code-start=0x080483a4 --trigger-code-stop=0x080483a4 –e BR_INST_RETIRED:taken - ./loop $i |
./save_output ; done
./analyze_simple
```

In this case, the two options of pfmon (-trigger-code-start and --trigger-code-stop) are used to specify the address range of the conditional branch instruction to be monitored.

A small program called *analyze_simple* is used to analyze the output of pfmon. It computes the difference between two consecutive results and checks if it is 2 or 1. Accordingly, when the difference equals 2, it outputs 1. On the other hand, when the difference equals 1, it produces 0. So, the sequence 1111011110 is retrieved in this case.

Attempts have been made on an Intel-Core based processer to try and get as much information about the key. The case on recovering the private key with such limited information is very tough since there is no information given about every branch done. A more general solution was proposed which is to detect the number of 1's in the private key by knowing the number of taken branches. An experiment with 100 different keys was made and it showed that the number of taken branches got each time is not much of a help since it is almost the same. That resulted that the number of taken branches will most lie between the ranges 366-368 taken branches. This range was achieved from the experiment regardless if the key had a high or low number of ones.

Finally, it has been analyzed that it is very unlikely to be able to retrieve the private key on such a processor. The same attack was possible on an Itanium 2 based processor which gave more information about the branches made.

5. Countermeasures

There are many *countermeasures* that can be used to protect RSA implementation of OpenSSL from SBPA attack. The simplest and most effective one involves sensitive processes to disable the access to the BTB unit. Nevertheless, this technique requires a new generation of processors. Another proposed technique is to remove all conditional branches from the sensitive code, and replace them with indirect branches that read the target address from the registers. Indeed, an indirect branch always causes a jump to the address read from the register. As a result, there is no prediction and thus the BTB will not be modified. Indirect branches are available in most architectures, including x86, IA64, MIPS and ARM, making the technique widely applicable [9].

6. Conclusion

Branch prediction is a recent method used to perform a side channel attack. In this paper a method on how branch prediction has been used to attack the OpenSSL implementation of RSA is shown. Branch prediction task is to predict where the next instruction is in the instruction stream. It provides two items: the direction of the branch, if it is taken or not and the target of the branch.

By analyzing the sequence of instructions that are executed by the RSA process, all the bits of the private key were successfully extracted. This attack was made on the modular exponentiation function of the RSA implementation which will give crucial information about the key. This kind of attack is possible when a spy process is set to monitor a victim process branching events and from that the information is extracted.

This attack was performed successfully on Intel Itanium processer and with certain limitation about the data that could be extracted on an Intel-Core based processor. As a conclusion, the later versions of OpenSSL correct this defect in the modular exponentiation by different techniques.

References

[1] Stinson, Douglas R. *Cryptography: Theory and Practice*. 3rd Ed. Chapman & Hall/CRC, New York, 2006.

[2] Gaddam, B. P., Mahat, P. R., and Namireddy, H. R. *Branch Prediction and Wrong Path Event Early Detection*. http://puspamahat.com/Documents/SOBP.doc

[3] Aciicmez, O., Seifert, J. P., and Koc, C. K. *Predicting Secret Keys via Branch Prediction.*

Cryptology ePrint Archive, Report 2006/288,2006. http://eprint.iacr.org

[4] Bidgoli, H. *Handbook of Information Security*. Vol. 3. John Wiley and Sons, California State University, 2006.

[5] Aciicmez, O., Koc, C. K., and Seifert, J. P. On the Power of Simple Branch Prediction. Cryptology ePrint Archive, Report 2006/351, 2006. http://eprint.iacr.org/.

[6] http://www.openssl.org/docs/

[7] Van Tilborg, H. *Encyclopedia of cryptography and security*. Springer, 2005.

[8] http://perfmon2.sourceforge.net/

[9] Schmidt, N., and Sliwowski, M. *Survey of Attempts at Cracking RSA Encryption*. Department ofComputer Science.The College of William & Mary. {fnjschm, marcing}@cs.wm.edu

Appendix A: Sample of pfmon Output

```
$ pfmon --long-smpl-periods=1 --smpl-entries=100 -e BRANCH_EVENT --
irange=0x4000000000000980-0x4000000000000990 -- ./loop 10


entry 0 PID:32006 TID:32006 CPU:2 STAMP:0x114d55ec72853e2 OVFL:4 LAST_VAL:1
SET:0 IIP:0x40000000000008c1  PMD8  : 0x4000000000000979 b=1 mp=0 bru=0 b1=1
valid=y
              source addr=0x4000000000000982
              taken=y prediction=success
      PMD9  : 0x40000000000009b2 b=0 mp=1 bru=0 b1=0 valid=y
              target addr=0x40000000000009b0
entry 1 PID:32006 TID:32006 CPU:2 STAMP:0x114d55ec728747a OVFL:4 LAST_VAL:1
SET:0 IIP:0x40000000000008c1  PMD8  : 0x4000000000000979 b=1 mp=0 bru=0 b1=1
valid=y
              source addr=0x4000000000000982
              taken=y prediction=success
      PMD9  : 0x40000000000009b2 b=0 mp=1 bru=0 b1=0 valid=y
              target addr=0x40000000000009b0
entry 2 PID:32006 TID:32006 CPU:2 STAMP:0x114d55ec7287b67 OVFL:4 LAST_VAL:1
SET:0 IIP:0x40000000000008c1  PMD8  : 0x4000000000000979 b=1 mp=0 bru=0 b1=1
valid=y
              source addr=0x4000000000000982
              taken=y prediction=success
      PMD9  : 0x40000000000009b2 b=0 mp=1 bru=0 b1=0 valid=y
              target addr=0x40000000000009b0
entry 3 PID:32006 TID:32006 CPU:2 STAMP:0x114d55ec7288259 OVFL:4 LAST_VAL:1
SET:0 IIP:0x40000000000008c1  PMD8  : 0x4000000000000979 b=1 mp=0 bru=0 b1=1
valid=y
              source addr=0x4000000000000982
              taken=y prediction=success
      PMD9  : 0x40000000000009b2 b=0 mp=1 bru=0 b1=0 valid=y
              target addr=0x40000000000009b0
entry 4 PID:32006 TID:32006 CPU:2 STAMP:0x114d55ec72888eb OVFL:4 LAST_VAL:1
SET:0 IIP:0x4000000000000890  PMD8  : 0x400000000000097f b=1 mp=1 bru=1 b1=1
valid=y
              source addr=0x4000000000000980
              taken=n prediction=FE failure
entry 5 PID:32006 TID:32006 CPU:2 STAMP:0x114d55ec7288f6f OVFL:4 LAST_VAL:1
SET:0 IIP:0x40000000000008c1  PMD8  : 0x4000000000000979 b=1 mp=0 bru=0 b1=1
valid=y
              source addr=0x4000000000000982
              taken=y prediction=success
      PMD9  : 0x40000000000009b2 b=0 mp=1 bru=0 b1=0 valid=y
              target addr=0x40000000000009b0
entry 6 PID:32006 TID:32006 CPU:2 STAMP:0x114d55ec72895e7 OVFL:4 LAST_VAL:1
SET:0 IIP:0x40000000000008c1  PMD8  : 0x4000000000000979 b=1 mp=0 bru=0 b1=1
valid=y
              source addr=0x4000000000000982
              taken=y prediction=success
      PMD9  : 0x40000000000009b2 b=0 mp=1 bru=0 b1=0 valid=y
              target addr=0x40000000000009b0
```

```
entry 7 PID:32006 TID:32006 CPU:2 STAMP:0x114d55ec7289c5f OVFL:4 LAST_VAL:1
SET:0 IIP:0x40000000000008c1  PMD8  : 0x4000000000000979 b=1 mp=0 bru=0 b1=1
valid=y
             source addr=0x4000000000000982
             taken=y prediction=success
      PMD9  : 0x40000000000009b2 b=0 mp=1 bru=0 b1=0 valid=y
             target addr=0x40000000000009b0
entry 8 PID:32006 TID:32006 CPU:2 STAMP:0x114d55ec728a2d0 OVFL:4 LAST_VAL:1
SET:0 IIP:0x40000000000009f0  PMD8  : 0x400000000000097b b=1 mp=1 bru=1 b1=1
valid=y
             source addr=0x4000000000000982
             taken=y prediction=FE failure
      PMD9  : 0x40000000000009b2 b=0 mp=1 bru=0 b1=0 valid=y
             target addr=0x40000000000009b0
entry 9 PID:32006 TID:32006 CPU:2 STAMP:0x114d55ec728a99f OVFL:4 LAST_VAL:1
SET:0 IIP:0x4000000000000a10  PMD8  : 0x400000000000097f b=1 mp=1 bru=1 b1=1
valid=y
             source addr=0x4000000000000980
             taken=n prediction=FE failure
```

Appendix B:

```
int BN_mod_exp_mont(BIGNUM *rr, const BIGNUM *a, const BIGNUM *p,
                    const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *in_mont)
        {
        ……..
        ……..

        start=1;  /* This is used to avoid multiplication etc
                   * when there is only the value '1' in the
                   * buffer. */
        wvalue=0;       /* The 'value' of the window */
        wstart=bits-1;  /* The top bit of the window */
        wend=0;         /* The bottom bit of the window */

        if (!BN_to_montgomery(r,BN_value_one(),mont,ctx)) goto err;
        for (;;)
                {
                if (BN_is_bit_set(p,wstart) == 0)
                        {
                        if (!start)
                                {
                                if (!BN_mod_mul_montgomery(r,r,r,mont,ctx))
                                goto err;
                                }
                        if (wstart == 0) break;
                        wstart--;
                        continue;
                        }
                /* We now have wstart on a 'set' bit, we now need to work out how bit a window to do.  To do
                this we need to scan forward until the last set bit before the end of the window */
                j=wstart;
                wvalue=1;
                wend=0;
                for (i=1; i<window; i++)
                        {
                        if (wstart-i < 0) break;
                        if (BN_is_bit_set(p,wstart-i))
                                {
                                wvalue<<=(i-wend);
                                wvalue|=1;
                                wend=i;
                                }
                        }
        …….
        …….
err:
        if ((in_mont == NULL) && (mont != NULL)) BN_MONT_CTX_free(mont);
        BN_CTX_end(ctx);
        bn_check_top(rr);
        return(ret);
        }
```